8
Princípios e Diretrizes para o Design Orientado a Aspectos

Este Capítulo apresenta alguns princípios para o design de sistemas orientados a aspectos e diretrizes preliminares para a modelagem orientada a aspectos usando `aSideML`.

O Design Orientado a Aspectos (DOA) é uma área relativamente nova, e ainda não surgiu um método para aplicação de elementos do modelo de aspectos tendo como base elementos do modelo de objetos.

No entanto, há alguns princípios bem definidos na engenharia de software, atualmente recomendados para o design de sistemas orientados a objetos, que podem ser afetados pelos conceitos e pela pragmática da POA.

Algumas propriedades e princípios relacionados à POA estão surgindo, aos poucos. Esse conjunto de princípios, convencionais ou emergentes, pode ser usado como um ponto de partida para a sugestão de diretrizes gerais para DOA.

As diretrizes apresentadas aqui surgiram de exemplos realizados usando aspectos e *crosscutting* para a implementação de diferentes tipos de aplicações, incluindo o SMA Portalware. Espera-se que essas soluções documentem experiência de modelagem que pode ser útil em diferentes categorias de domínios e que facilite a implementação usando linguagens orientadas a aspectos. As diretrizes são explicadas em texto simples e ilustradas usando a `aSideML`.

8.1
Princípios

O design de software é repleto de princípios e técnicas para o gerenciamento de *dependências entre módulos* com o objetivo de promover a fácil reutilização e a evolução. Alguns princípios gerais são apresentados na Tabela 8.1, e os princípios usados no design orientado a objetos são apresentados na Tabela 8.2.

PUC-Rio - Certificação Digital Nº 9816120/CA

Princípio	Definição
Separação de *concerns*	*Toda questão importante (ou* concern*) deve ser isoladamente considerado* [36].
Acoplamento baixo	*Todo módulo deve se comunicar com o menor número possível de módulos* [100].
Acoplamento fraco	*Se dois módulos se comunicam, eles devem trocar a menor quantidade de informação possível* [100].
Ocultamento de Informação	*Todas as informações sobre um componente devem ser privativas a esse componente a menos que sejam especificamente declaradas públicas* [100].
Coesão	*Componentes logicamente relacionados devem ser agrupados* [39].

Tabela 8.1: Princípios Gerais.

Tabela 8.2: princípios relacionados à orientação a objetos

Princípio	Definição
Open Closed Principle (OCP)	*Um módulo deve estar aberto para extensão porém fechado para modificação* [94].
Liskov Substitution Principle (LSP)	*Uma classe derivada pode substituir uma superclasse (classe base)* [85].
Dependency Inversion Principle (DIP)	*Dependa de abstrações; não dependa de elementos concretos* [94].
Interface Segregation Principle (ISP)	*Prover diversas interfaces, uma para cada tipo de cliente, ao invés de uma interface única, de propósito geral* [94].
Lei de Demeter (LdD)	*Cada unidade deve conhecer apenas unidades intimamente relacionadas a ela* [78].

Tabela 8.2: Princípios relacionados à orientação a objetos.

PUC-Rio - Certificação Digital Nº 9816120/CA

O design estruturado exibe um tipo particular de estrutura de dependência, o *top-down*, que a partir do topo, aponta na direção dos detalhes. Os *módulos de nível mais alto dependem dos módulos de nível inferior*, que dependem dos módulos de nível mais inferior ainda.

O design orientado a objetos apresenta uma estrutura de dependência na qual a *maioria das dependências aponta para as abstrações*. Além do mais, não se depende mais dos módulos que contêm a implementação detalhada, e sim estes dependem das abstrações. Portanto, a estrutura de dependência fornecida pelo design estruturado foi invertida [94].

As tecnologias de POA interferem com esses princípios ao fornecer um novo tipo de *dependência modular*: a dependência entre aspectos e componentes.

8.1.1
Princípios de Design Orientado a Aspectos

O Design Orientado a Aspectos (DOA) pode ser definido como um método de design que oferece suporte à separação de *concerns* através de duas dimensões de decomposição: uma dimensão baseada em componentes e uma dimensão baseada em aspectos. DOA incorpora o processo de *decomposição orientado a aspectos* além da decomposição algorítmica e orientada a objetos. Ademais, algumas técnicas de POA (como AspectJ) reforçam a *dicotomia aspecto-base*, a *quantificação* e a *transparência* dos componentes (consulte o Capítulo 3). Essas propriedades também têm o suporte de DOA. A decomposição orientada a aspectos resulta em uma estrutura de dependência na qual as dependências começam nos módulos de aspectos e dirigem-se para os componentes (Figura 8.1).

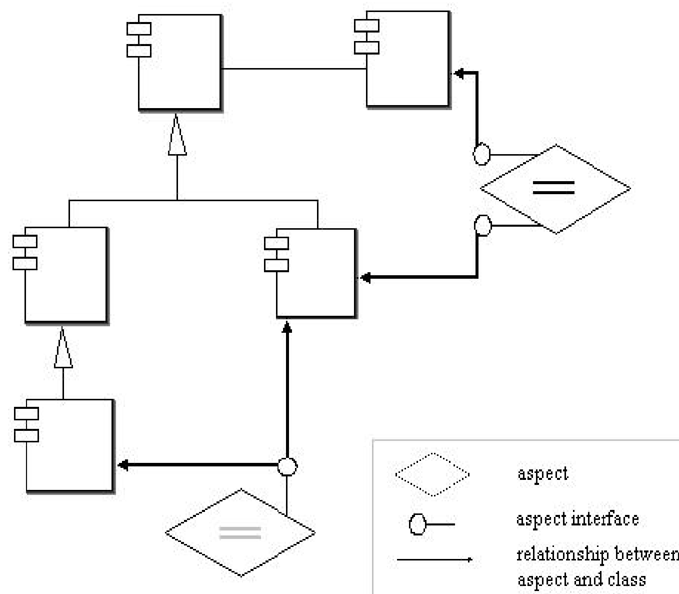

Figura 8.1: Dependências no design orientado a aspectos.

Nessa Seção, fornecemos nossa visão preliminar sobre a forma como as técnicas da POA interferem nos princípios de design existentes e seu potencial para reduzir ou melhorar as dependências em um sistema, a fim de caracterizar o DOA.

Acoplamento A POA oferece suporte a um baixo acoplamento entre componentes. Os aspectos dependem dos componentes e os componentes podem não depender dos aspectos quando há a propriedade de transparência. Os aspectos modularizam características que, de outra forma, ficariam entrelaçadas ou espalhadas com vários componentes. P

PUC-Rio - Certificação Digital Nº 9816120/CA

ortanto, o uso de aspectos pode minimizar as dependências entre componentes. O baixo acoplamento entre aspectos também é necessário. Na prática, quando vários aspectos afetam o mesmo modelo, eles podem ser acoplados entre si de várias maneiras (Seção 4.2.1).

Coesão A POA oferece suporte a módulos mais coesos. A coesão é uma medida da ligação entre os elementos de um único módulo. Os aspectos localizam os elementos relacionados a *concerns* que ficariam entrelaçados ou espalhados com vários módulos de componente; esses módulos tendem a ser mais coesos [39].

Ocultamento de Informação Sempre que possível, a POA permite o ocultamento de informação. O ocultamento de informação (também conhecidao como encapsulamento) recebeu um interesse considerável no contexto da POO. Em particular, o encapsulamento de objetos compostos por herança foi discutido repetidas vezes [123]. Algumas abordagens à POA oferecem meios para quebrar o encapsulamento da implementação de *concerns* e permitem que os aspectos manipulem características particulares das classes (*private*). Por outro lado, os aspectos encapsulam e localizam características que, de outra forma, ficaram entrelaçadas ou espalhadas em vários componentes.

OCP A POA garante o OCP. O princípio OCP ajuda os projetistas a criarem módulos extensíveis, sem precisarem ser alterados. Isso significa que podem ser adicionadas novas características ao código existente, sem alterá-lo, apenas adicionando novo código. Com a POA, as classes base são a parte fechada e os aspectos são a parte aberta [104].

LSP A POA pode ajudar a garantir o LSP entre as classes. O princípio LSP ajuda o projetista a garantir a *herança estrita*, isto é, a herança que prescreve a compatibilidade do comportamento entre classes e suas subclasses. Esse princípio vai além da simples verificação de assinaturas, levando em consideração o comportamento das operações (conforme definido pelas precondições e pós-condições) e as invariantes definidas pelas classes. Aspectos podem localizar redefinições e adições que poderiam quebrar o LSP, que, de outra forma, se organizariam em subclasses. Contudo, os clientes podem ter seu comportamento estendido a fim de lidar com os objetos estendidos por esses aspectos. Os aspectos também são úteis para modularizar as precondições e pós-condições que devem compor as operações de classes.

PUC-Rio - Certificação Digital Nº 9816120/CA

DIP A POA também procura garantir o DIP. O princípio DIP orienta o projetista para depender de interfaces e classes abstratas, uma vez que tudo o que é concreto muda com mais freqüência. Nenhuma dependência deveria ser direcionada para uma classe concreta. Os aspectos devem ser compostos o mais alto possível em uma hierarquia de classes [104]. Em outra dimensão, DIP relaciona-se a SDP, o *Stable Dependencies Principle* (Princípio de Dependências Estáveis) [94]: "Dependa na direção da estabilidade". Se considerarmos que as classes funcionais são mais estáveis do que aspectos, então SDP prescreve a dependência de aspectos sobre classes.

ISP A POA também pode ajudar a garantir o ISP. O princípio ISP afirma que, se uma classe possui vários clientes, em vez de uma sobrecarga nas classes com todos os métodos de que os clientes precisam, o projetista deve criar interfaces específicas para cada cliente e fazer as classes herdarem de todas as interfaces necessárias. Os aspectos oferecem suporte à separação e à modularização de visões que podem ser combinadas em várias classes.

LdD A *Lei de Demeter* (LdD) é um princípio de design que afirma que um método deve conter apenas mensagens enviadas a si mesmo, a variáveis de instâncias locais e/ou argumentos de métodos (veja Seção 2.3.3). Técnicas como a da Programação Adaptativa [78] trata das consequências de aplicar a LdD, e desse modo, promove a adoção dessa lei.

PUC-Rio - Certificação Digital Nº 9816120/CA

A medição das propriedades estruturais do design de artefatos de software, como acoplamento, coesão e separação de *concerns* e outros originados dos princípios listados anteriormente, é um caminho promissor na direção das primeiras avaliações de qualidade de sistemas de POA.

Em [118], apresentamos um framework, com base em um conjunto de métricas e um modelo de qualidade, para ajudar na avaliação de softwares orientados a aspectos em termos de reusabilidade e manutenabilidade, que trata das novas abstrações e das novas dimensões de acoplamento e coesão introduzidas pelo DOA. Ainda são necessárias mais investigações experimentais para avaliar a forma como a POA interfere com outros princípios existentes.

8.2 Diretrizes

Aspectos são usados para resolver vários problemas. Na Seção 4.1, organizamos esses aspectos em algumas categorias, não necessariamente disjuntas: aspectos sistêmicos, colaborativos, subjetivos e evolutivos. Cada categoria coloca alguns desafios específicos à prática atual da modelagem de software que podem ser resolvidos pela modelagem orientada a aspectos.

Nesta Seção, apresentamos algumas regras e diretrizes para a MOA e usamos diagramas de `aSideML` para ilustrá-las. Essas diretrizes podem forçar um ou mais princípios de DOA discutidos anteriormente.

8.2.1 Regra Geral para DOA

> *Tente manter um modelo de objetos independente, que possa ser estendido por aspectos* [69].

A adoção dessa regra geral respeita a *dicotomia aspecto-base* e a *transparência*. A adoção dessa regra facilitou a construção e a evolução do SMA Portalware apresentado no Capítulo 7.

8.2.2 Aspectos e Transparência

Aspectos podem adicionar características transversais a classes base que podem ser explicitamente usadas por outras classes ou até mesmo por aspectos. Nesses casos, as classes e os aspectos não são independentes em relação ao aspecto anterior e a transparência diminui. Apresentamos duas soluções orientadas a aspectos para lidar com o mesmo problema: a dependência indesejável entre classes e aspectos sempre que uma chamada explícita a um método introduzido é feita a partir de uma classe. A operação introduzida não deve ser explicitamente invocada a partir de outra classe. A solução proposta reforça o baixo acoplamento e a transparência.

Primeiro Cenário

A classe base estendida pela adição também possui um método que pode ser estendido por uma operação de *crosscutting* que contém uma chamada explícita ao novo método.

PUC-Rio - Certificação Digital Nº 9816120/CA

Intenção. Minimizar as dependências entre classes e aspectos; evitar chamadas explícitas a métodos introduzidos a partir de classes base.

Solução. Introduzir outras operações em uma classe e afetar um método original na *mesma* classe. Usar uma interface transversal a fim de modularizar a adição e o refinamento.

Exemplo.

Consulte o aspecto `Adaptation` apresentado na Figura 7.18. A operação `comparePlan` é introduzida na classe `Agent`, e o advice `adaptPlan_` afeta a operação `setGoal` de `Agent`. A chamada explícita a `comparePlan` é definida dentro do corpo do advice. Remover o aspecto do modelo não deixa inconsistência.

Estrutura.

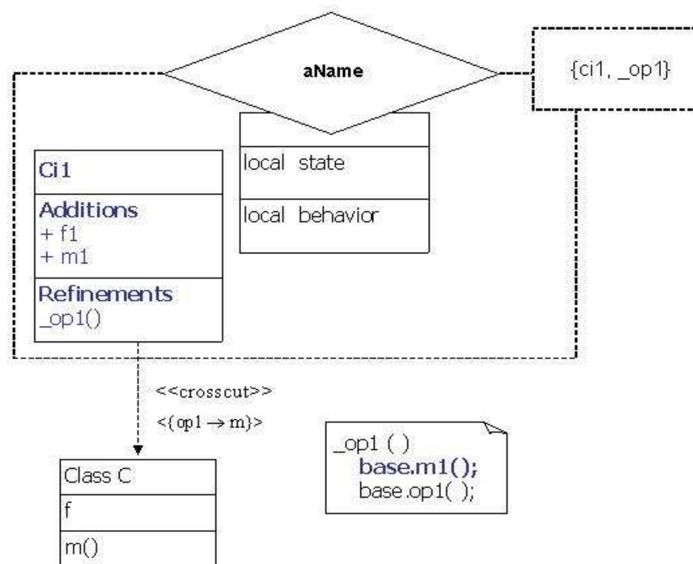

Figura 8.2: Refinamento e adição de operação usando uma interface transversal.

Segundo Cenário

A classe base que é estendida pela adição, e outra classe que possui um método que pode ser estendido por uma operação de *crosscutting* ; esse método contém uma chamada explícita ao novo método.

Intenção. Minimizar as dependências entre classes e aspectos; evitar chamadas explícitas a métodos introduzidos a partir de classes base.

PUC-Rio - Certificação Digital Nº 9816120/CA

Solução. Introduzir outra operação em uma classe e usá-la a partir de *outra* classe. Usar duas interfaces transversais.

Descrição.

A solução proposta é definir um advice que afeta um método existente `m2` da classe `C2` e codificar a chamada explícita dentro do corpo do advice. Como o uso efetivo desse advice depende da introdução anterior, deve ser definido um aspecto com duas interfaces transversais, uma para a introdução da nova operação `n` na classe `C1` e outra para oferecer um advice a alguma operação `m2` da classe `C2` para chamar `n`. Deve estar disponível para `m2` uma referência para uma instância da classe `C1`, provavelmente como um parâmetro ou membro de objeto.

Exemplo.

Consulte as classes `Sensor` e `Agent` e o aspecto `Interaction` na Figura 7.14. O método `receiveMsg` é introduzido na classe `Agent` por meio da interface transversal `IMessaging` e o método `registerMessage` é afetado pelo advice `incomingMsg`. O código do advice contém uma chamada explícita a `agent.receiveMgs()`.

Estrutura.

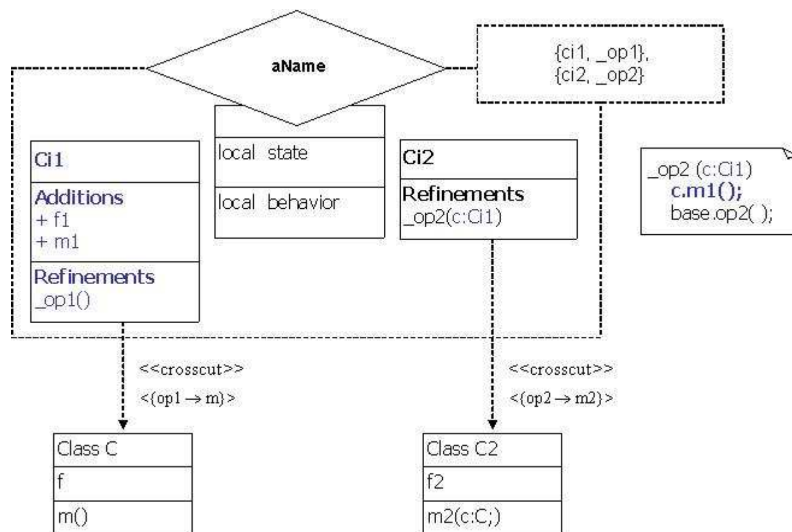

Figura 8.3: Refinamento e introdução de operação (duas interfaces transversais).

PUC-Rio - Certificação Digital Nº 9816120/CA

8.2.3
Aspectos para Colaboração

Uma colaboração em um programa orientado a objetos costuma envolver diferentes classes que colaboram. Todavia, como as colaborações não são cidadãos de primeira classe no design orientado a objetos, elas acabam localizadas em um método em uma das classes ou são divididas em métodos em cada uma das classes envolvidas. A última solução espalha a colaboração através de várias classes, dificultando a adaptação quando há alterações na colaboração.

Aspectos podem ser usados para localizar colaborações entre as classes participantes. Nesse caso, os aspectos afetam as classes de maneira heterogênea; portanto, requerem suporte a *crosscutting* horizontal. Entretanto, não é possível expressar de maneira clara essas facetas sem o suporte de interfaces transversais.

Intenção. Localizar uma colaboração entre classes e enfatizar os papéis participantes.

Solução. Definir um aspecto com uma interface transversal para cada papel participante. O aspecto localiza as interações entre as classes participantes e as regras que regem essas interações. As interfaces transversais localizam o comportamento relacionado ao tipo de participante.

Estrutura.

A Figura 8.4 ilustra a solução.

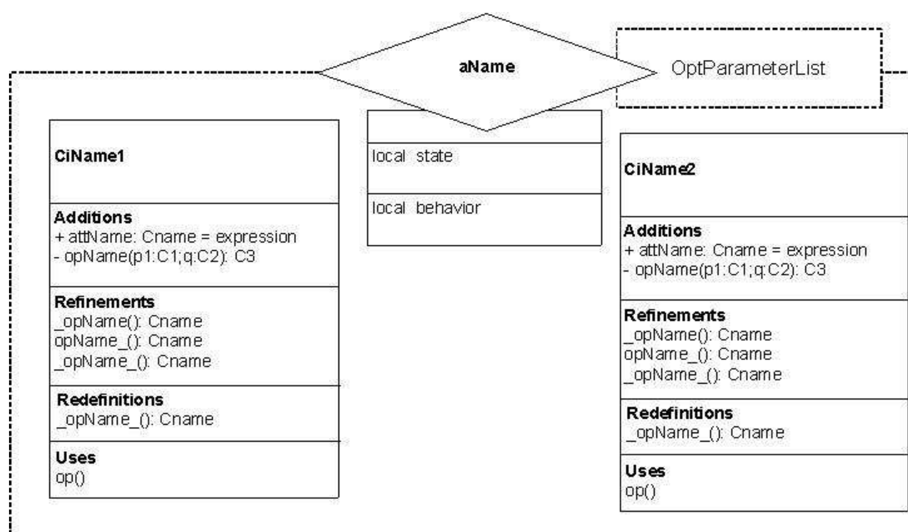

Figura 8.4: Aspecto para Colaboração.

PUC-Rio - Certificação Digital Nº 9816120/CA

Descrição.

O aspecto nomeia um conjunto de pontos de interação em suas interfaces transversais. Cada interface transversal descreve uma classe participante separada e pode estendê-la a fim de exercer um papel específico na colaboração. O comportamento e o estado local do aspecto são usados para especificar o protocolo de colaboração.

Aspectos para colaboração podem agir como conectores de software e inserir estrutura e comportamento de forma precisa e não-antecipada nas partes que conectam. Referimo-nos a esse tipo de *crosscutting* como a colaboração ou a *dimensão horizontal* de *crosscutting* .

Exemplo.

O sistema `Telecom` (c.f. [76]) contém um modelo simples de conexões de telefone às quais sincronização (*timing*) e faturamento (*billing*) são adicionados usando aspectos. O *concern* de *Timing* está relacionado à sincronização das conexões entre clientes e à manutenção do tempo total de conexão por cliente. Se uma conexão estiver completa, o cronômetro (*timer*) deve ser ativado. Quando uma conexão é interrompida, o cronômetro deve ser parado. O tempo consumido durante uma conexão deve ser associado ao pagador (o cliente que fez a ligação). A Figura 8.5 apresenta o aspecto `Timing`.

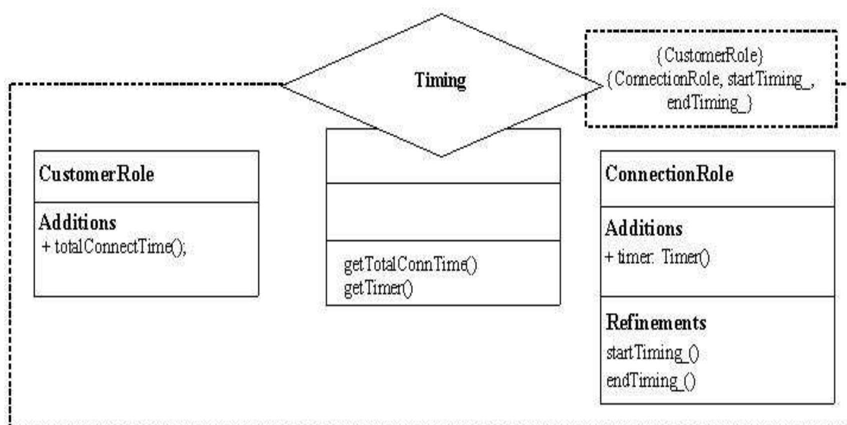

Figura 8.5: O aspecto `Timing`.

8.2.4 Aspectos para Evolução

Intenção Evoluir uma classe base de forma que ofereça novos serviços ou seja capaz de exercer um novo papel, sem alterações invasivas.

PUC-Rio - Certificação Digital Nº 9816120/CA

Estrutura.

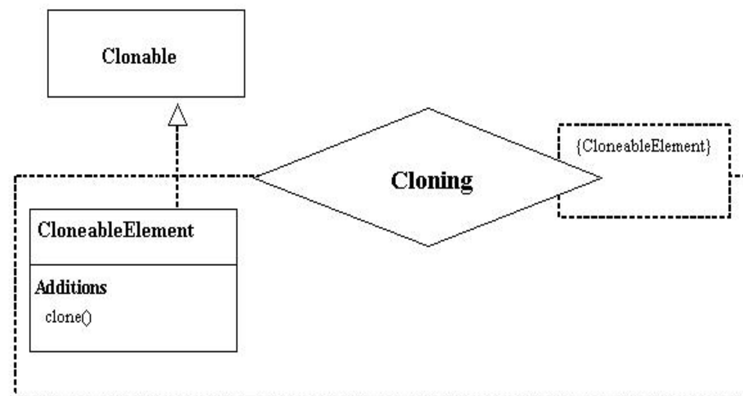

Figura 8.6: Aspecto para evolução.

Solução. Definir um aspecto com uma interface transversal que implementa uma interface ou estende uma classe. Usar o compartimento Additions a fim de declarar os atributos e métodos que serão implementados.

Exemplo. A Figura 8.6 ilustra um aspecto que torna instâncias de `Point` elementos clonáveis [10]. O aspecto afetará classes denotadas por `CloneableElement`.

8.2.5 Aspectos para Visões

Uma classe pode ter muitos clientes. Em vez de definir uma grande interface geral para servir a todos, as necessidades dos clientes podem ser localizadas em visões menores e separadas, específicas a cada tipo de cliente. Essas visões são associadas a uma classe base e são implementadas nela. Aspectos subjetivos podem ser usados para tratar da necessidade dessas *visões segregadas*, reforçando o uso do princípio ISP.

Aspectos podem ser usados para oferecer suporte à segregação de interface da seguinte forma:

– Uma classe oferece uma visão padrão disponível para cada cliente.

– Um aspecto localiza novos métodos e atributos *privados* que correspondem a uma nova visão usando uma interface transversal. Esses elementos são introduzidos na classe base (a classe servidor).

PUC-Rio - Certificação Digital Nº 9816120/CA

– O mesmo aspecto, por meio de outra interface transversal, afeta outras classes (os clientes) com outros comportamentos que permitem que eles (e somente eles) usem serviços privados introduzidos na classe servidor.

Aspectos subjetivos também oferecem suporte a extensões de comportamento dependentes do contexto, como a necessidade de registrar, monitorar ou dar acesso a algum comportamento somente quando observado sob uma determinada perspectiva ou invocado por um determinado cliente. Aspectos podem afetar apenas chamadas de métodos selecionados, dependendo do tipo de cliente e oferecem outros comportamentos, customizáveis por cliente.

8.3 Trabalhos Relacionados

A comunidade de AspectJ ofereceu muitas contribuições na forma de idiomas, receitas, regras e padrões para a linguagem AspectJ [7, 55, 56, 57, 58]. Elas descrevem soluções que se mostraram úteis na construção de sistemas com AspectJ.

Alguns padrões também foram propostos para *concerns* e domínios específicos como sistemas multiagentes [54], distribuição [125], etc. Essas soluções baseiam-se na experiência e devem crescer com o desenvolvimento de novos sistemas orientados a aspectos.

8.4 Considerações Finais

Neste Capítulo, avaliamos os princípios do DOA e documentamos algumas lições aprendidas durante o desenvolvimento e a avaliação de aplicações da POA, incluindo o SMA Portalware. Trabalhos em desenvolvimento incluem a documentação de outros usos comuns de aspectos no nível do design com `aSideML` (aspectos para garantia de contratos, extensão incremental etc.) e a avaliação de como interferem com os princípios usados no design orientado a objetos.

Como a POA é uma disciplina recente, com foco claro em problemas de implementação e estilo de programação e, naturalmente, com poucas diretrizes disponíveis para o suporte à modelagem e ao design orientado a aspectos, nossa avaliação informal dos princípios e o conjunto inicial de diretrizes também constituem uma contribuição deste trabalho.

PUC-Rio - Certificação Digital Nº 9816120/CA